# FUNCTIONAL AND CONCURRENT PROGRAMMING

*Core Concepts and Features*

MICHEL
**CHARPENTIER**

*Foreword by*
**CAY HORSTMANN**

# *Functional and Concurrent Programming*

*This page intentionally left blank*

# Functional and Concurrent Programming

## Core Concepts and Features

Michel Charpentier

Addison-Wesley
Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover Image: echo3005 / Shutterstock

Library of Congress Number: 2022946382
Copyright © 2023 Pearson Education, Inc.

ISBN-13: 978-0-13-746654-2
ISBN-10: 0-13-746654-4

ScoutAutomatedPrintCode

To Karen and Andre

# Contents

*This page intentionally left blank*

# List of Listings

# List of Figures

# Foreword by Cay Horstmann

In my book *Scala for the Impatient*, I provide a rapid-fire introduction into the many features of the Scala language and API. If you need to know how a particular feature works, you will find a concise explanation and a minimal code example (with real code, not fruits or animals). I assume that the reader is familiar with Java or a similar object-oriented programming language and organize the material to maximize the experience and intuition of such readers. In fact, I wrote the book because I was put off by the learning materials at the time, which were disdainful of object-oriented programming and biased toward functional programming as the superior paradigm.

That was more than a decade ago. Nowadays, functional techniques have become much more mainstream, and it is widely recognized that the object-oriented and functional paradigms complement each other. In this book, Michel Charpentier provides an accessible introduction to functional and concurrent programming. Unlike my Scala book, the material here is organized around concepts and techniques rather than language features. Those concepts are developed quite a bit more deeply than they would be in a book that is focused on a programming language. You will learn about nontrivial and elegant techniques such as zippers and trampolines.

This book uses Scala 3 for most of its examples, which is a great choice. The concise and elegant Scala syntax makes the concepts stand out without being obscured by a thicket of notation. You will particularly notice that when the same concept is expressed in Scala and in Java. You don't need to know any Scala to get started, and only a modest part of Scala is used in the code examples. Again, the focus of the book is concepts, not programming language minutiae. Mastering these concepts will make you a better programmer in any language, even if you never end up using Scala in your career.

I encourage you to actively work with the sample programs. Execute them, observe their behavior, and experiment by making changes. I suggest that you use a programming environment that supports Scala worksheets, such as Visual Studio Code, IntelliJ, or the online Scastie service. With a worksheet, turnaround is quick and exploratory programming is enjoyable.

Seven out of the 28 chapters are complete case studies that illustrate the material that preceded them. They are chosen to be interesting without being overwhelming. I am sure you will profit from working through them in detail.

The book is divided into two parts. The first part covers functional programming with immutable data, algebraic data types, recursion, higher-order functions, and lazy evaluation. Even if you are at first unexcited about reimplementing lists and trees, give it a chance. Observe the contrast with traditional mutable data structures, and you will find the journey rewarding. The book is blessedly free of complex category theory that in my opinion—evidently shared by the author—requires a large amount of jargon before yielding paltry gains.

The focus of the second part is concurrent programming. Here too the organization along concepts rather than language and API features is refreshing. Concurrent programming is a complex subject with many distinct use cases and no obvious way of teaching it well. Michel has broken down the material into an interesting and thought-provoking sequence of chapters that is quite different from what you may have seen before. As with the first part, the ultimate aim is not to teach you a specific set of skills and techniques, but to make you think at a higher level about program design.

I enjoyed reading and working through this unique book and very much hope that you will too.

Cay Horstmann
Berlin, 2022

# Preface

Before you start reading this book, it is important to think about the distinction between programming *languages* and programming language *features*. I believe that developers benefit from being able to rely on an extensive set of programming language features, and that a solid understanding of these features—in *any* language—will help them be productive in a variety of programming languages, present or future.

The world of programming languages is varied and continues to evolve all the time. As a developer, you are expected to adapt and to repeatedly transfer your programming skills from one language to another. Learning new programming languages is made easier by mastering a set of core features that today's languages often share, and that many of tomorrow's languages are likely to use as well.

Programming language features are illustrated in this book with numerous code examples, primarily in Scala (for reasons that are detailed later). The concepts, however, are relevant—with various degrees—to other popular languages like Java, C++, Kotlin, Python, C#, Swift, Rust, Go, JavaScript, and whatever languages might pop up in the future to support strong typing as well as functional and/or concurrent programming.

As an illustration of the distinction between languages and features, consider the following programming task:

> *Shift every number from a given list by a random amount between* `-10` *and* `10`*. Return a list of shifted numbers, omitting all values that are not positive.*

A Java programmer might implement the desired function as follows:

*Java*

```java
List<Integer> randShift(List<Integer> nums, Random rand) {
  var shiftedNums = new java.util.ArrayList<Integer>(nums.size());
  for (int num : nums) {
    int shifted = num + rand.nextInt(-10, 11);
    if (shifted > 0) shiftedNums.add(shifted);
  }
  return shiftedNums;
}
```

A Python programmer might write this instead:

Python
```
def rand_shift(nums, rand):
    shifted_nums = []
    for num in nums:
        shifted = num + rand.randrange(-10, 11)
        if shifted > 0:
            shifted_nums.append(shifted)
    return shifted_nums
```

Although they are written in two different languages, both functions follow a similar strategy: Create a new empty list to hold the shifted numbers, shift each original number by a random amount, and add the new values to the result list only when they are positive. For all intents and purposes, the two programs are the same.

Other programmers might choose to approach the problem differently. Here is one possible Java variant:

Java
```
List<Integer> randShift(List<Integer> nums, Random rand) {
  return nums.stream()
      .map(num -> num + rand.nextInt(-10, 11))
      .filter(shifted -> shifted > 0)
      .toList();
}
```

The details of this implementation are not important for now—it relies on functional programming concepts that will be discussed in Part I. What matters is that the code is noticeably different from the previous Java implementation.

You can write a similar functional variant in Python:

Python
```
def rand_shift(nums, rand):
    return list(filter(lambda shifted: shifted > 0,
                       map(lambda num: num + rand.randrange(-10, 11), nums)))
```

This implementation is arguably closer to the second Java variant than it is to the first Python program.

These four programs demonstrate two different ways to solve the original problem. They contrast an imperative implementation—in Java or in Python—with a functional implementation—again, in Java or in Python. What fundamentally distinguishes the programs is not the languages—Java versus Python—but the features being used—imperative versus functional. The programming language features used in the imperative variant (assignment statements, loops) and in the functional variant (higher-order functions, lambda expressions) exist independently from Java and Python; indeed, they are available in many programming languages.

I am not saying that programming languages don't matter. We all know that, for a given task, some languages are a better fit than others. But I want to emphasize core features and concepts that extend across languages, even when they appear under a different syntax. For instance, an experienced Python programmer is more likely to write the example functional program in this way:

*Python*

```python
def rand_shift(nums, rand):
    return [shifted for shifted in (num + rand.randrange(-10, 11) for num in nums)
            if shifted > 0]
```

This code looks different from the earlier Python code—and the details are again unimportant. Notice that functions `map` and `filter` are nowhere to be seen. Conceptually, though, this is the same program but written using a specific Python syntax known as list comprehension, instead of `map` and `filter`.

The important concept to understand here is the use of `map` and `filter` (and more generally higher-order functions, of which they are an example), not list comprehension. You benefit from this understanding in two ways. First, more languages support higher-order functions than have a comprehension syntax. If you are programming in Java, for instance, you will have to write `map` and `filter` explicitly (at least for now). Second, if you ever face a language that uses a somewhat unusual syntax, as Python does with list comprehension, it will be easier to recognize what is going on once you realize that it is just a variation of a concept you already understand.

The preceding code examples illustrate a contrast between a program written in plain imperative style and one that leverages the functional programming features available in many languages. I can make a similar argument with concurrent programming. Languages (and libraries) have evolved, and there is no reason to write today's concurrent programs the way we did 20 years ago. As a somewhat extreme example, travel back not quite 20 years to 2004, the days of Java 1.4, and consider the following problem:

> *Given two tasks that each produce a string, invoke both tasks in parallel and return the first string that is produced.*

Assume a type `StringComputation` with a string-producing method `compute`. In Java 1.4, the problem can be solved as follows (do not try to understand the code; it is rather long, and the details are unimportant):

Java
```java
String firstOf(final StringComputation comp1, final StringComputation comp2)
    throws InterruptedException {
  class Result {
    private String value = null;

    public synchronized void setValue(String str) {
      if (value == null) {
        value = str;
        notifyAll();
      }
    }

    public synchronized String getValue() throws InterruptedException {
      while (value == null)
        wait();
      return value;
    }
  }

  final Result result = new Result();
  Runnable task1 = new Runnable() {
    public void run() {
      result.setValue(comp1.compute());
    }
  };
  Runnable task2 = new Runnable() {
    public void run() {
      result.setValue(comp2.compute());
    }
  };
  new Thread(task1).start();
  new Thread(task2).start();
  return result.getValue();
}
```

This implementation uses features with which you may not be familiar (but which are covered in Part II of the book).[1] Here are the important points to notice:

- The code is about 30 lines long.
- It relies on *synchronized* methods, a form of locking available in the Java Virtual Machine (JVM).

[1]One reason such old-fashioned features are still covered in this book is that I believe they help us understand the richer and fancier constructs that we should be using in practice. The other reason is that the concurrent programming landscape is still evolving and recent developments, such as virtual threads in the Java Virtual Machine, have the potential to make these older concepts relevant again.

- It uses methods `wait` and `notifyAll`, which implement a basic synchronization scheme on the JVM.
- It starts its own two threads to run the two tasks in parallel.

Fast forward to today's Java, and reimplement the program:

*Java*

```java
String firstOf(StringComputation comp1, StringComputation comp2, Executor threads)
    throws InterruptedException, ExecutionException {
  var result = new CompletableFuture<String>();
  result.completeAsync(comp1::compute, threads);
  result.completeAsync(comp2::compute, threads);
  return result.get();
}
```

Again, ignore the details and observe these points:

- The code is much shorter.
- Class `Result` is gone. It implemented a poor man's form of a *future*, but futures are now available is many languages, including Java.
- Synchronized methods are gone. The code does not rely on locks anywhere.
- Methods `wait` and `notifyAll` are gone. Instead, `CompletableFuture` implements its own synchronization, correctly and efficiently.
- No thread is created explicitly. Instead, threads are passed as an argument in the form of an `Executor` and can be shared with the rest of the application.

There is one more difference between the two variants that I want to emphasize. In the newer code, the two `Runnable` classes have disappeared. They have been replaced with an odd-looking syntax that did not exist in Java 1.4: `comp1::compute`. You may find this syntax puzzling because method `compute` seems to be missing its parentheses. Indeed, this code does not invoke `compute`, but rather uses the method itself as an argument to `completeAsync`. It could be written as a lambda expression instead: `comp1::compute` is the same as `() -> comp1.compute()`. Passing functions as arguments to functions is a fundamental concept of functional programming, which is explored at length in Part I, but finds frequent uses in writing concurrent code as well.

Here's the point of this illustration: You can still write the first version of the program in today's Java, but you shouldn't. It is notoriously difficult to get multithreaded code correct, and it is even more difficult to get it correct *and* make it efficient. Instead, you should leverage what is available in the language and use it effectively. Are you making the most of the programming languages you are using today?

As a trend, programming languages have become more abstract and richer in features, a shift that makes many programming tasks less demanding. There are more concepts to understand in Java 19 than there were in Java 1, but it is easier to write correct and efficient programs with Java 19 than it was with Java 1. Feature-rich programming languages can be harder to learn, but they are also more powerful once mastered.

Of course, what you find hard or easy depends a lot on your programming background, and it is important not to confuse simplicity with familiarity. The functional variants of the Java and Python programs presented earlier are not more complicated than the imperative variants, but for some programmers, they can certainly be less familiar. Indeed, it is more difficult for a programmer to shift from an imperative to a functional variant (or vice versa) within Java or Python than it is to shift from Java to Python (or vice versa) within the same imperative or functional style. The latter transition is mostly a matter of syntax, while the first requires a paradigm shift.

Most of the advantages of current, feature-rich, programming languages revolve around functional programming, concurrency, and types—hence the three themes of this book. A common trend is to provide developers with abstractions that allow them to dispense with writing nonessential implementation details, and code that is not written is bug-free code.

Jumps and gotos, for instance, were long ago discarded in high-level programming languages in favor of structured loops. But many loops can themselves be replaced with functional alternatives that instead use a standard set of higher-order functions. Similarly, writing concurrent programs directly in terms of threads and locks can be very challenging. Relying on thread pools, futures, and other mechanisms instead can result in simpler patterns. In many scenarios, you have no more reason to use loops and locks than you have to write your own hash map or sorting method: It's unnecessary work, it's error-prone, and it's unlikely to achieve the performance of existing implementations. As for types, the age-old dichotomy between safety—being able to catch errors thanks to types—and flexibility—not being overly constrained in design choices because of types—is often being resolved in favor of safe *and* flexible type systems, albeit complicated ones.

This book is not a comprehensive guide to everything you need to know about functional and concurrent programming, or about types. But to leverage modern language constructs in your everyday programming, you need to become familiar with the abstract concepts that underlie these features. There is more to applying functional patterns than being aware of the syntax for lambda expressions, for instance. This book introduces only enough concepts as are needed to use language features effectively. There is a lot more to functional and concurrent programming and to types than what the book covers. (There is also a lot more to Scala.) Advanced topics are left for you to explore through other resources.

## Why Scala?

As mentioned earlier, most of the code illustrations in this book are written in Scala. This may not be the language you are most familiar with, or the language in which you plan to develop your next application. It is a fair question to wonder why I chose it instead of a more mainstream language.

Scala is a programming language that aims to combine object-oriented and functional programming, with good support for concurrency as well.[2] It is a hybrid language—also called a multi-paradigm language. In fact, all three versions of the random shifting program already written in Java and in Python can be written in Scala:

Scala

```
def randShift(nums: List[Int], rand: Random): List[Int] = {
   val shiftedNums = List.newBuilder[Int]
   for (num <- nums) {
      val shifted = num + rand.between(-10, 11)
      if (shifted > 0) {
         shiftedNums += shifted
      }
   }
   shiftedNums.result()
}

def randShift(nums: List[Int], rand: Random): List[Int] =
   nums.view
      .map(num => num + rand.between(-10, 11))
      .filter(shifted => shifted > 0)
      .toList

def randShift(nums: List[Int], rand: Random): List[Int] =
   for {
      num <- nums
      shifted = num + rand.between(-10, 11)
      if shifted > 0
   } yield shifted
```

The first function is imperative, based on an iteration and a mutable list. The next variant is functional and uses `map` and `filter` explicitly. The last variant relies on Scala's `for`-comprehension, a mechanism similar to (but more powerful than) Python's list comprehension.

You can also use Scala to write a concise solution to the concurrency problem. It uses futures and thread pools, like the earlier Java program:

Scala

```
def firstOf(comp1: StringComputation, comp2: StringComputation)
           (using ExecutionContext): String = {
   val future1 = Future(comp1.compute())
   val future2 = Future(comp2.compute())
   Await.result(Future.firstCompletedOf(Set(future1, future2)), timeout)
}
```

[2]Different incarnations of Scala exist. This book uses the most common flavor of Scala, namely, the one that runs on the JVM and leverages the JVM's support for concurrency.

Given the book's objectives, there are several benefits to using Scala for code illustrations. First, this language is feature-rich, making it possible to illustrate many concepts without switching languages. Many of the standard features of functional and concurrent programming exist in Scala, which also has a powerful type system. Second, Scala was introduced fairly recently and was carefully (and often beautifully) designed. Compared to some older languages, there is less historical baggage in Scala that can get in the way when discussing underlying concepts. Finally, Scala syntax is quite conventional and easy to follow for most programmers without prior exposure to the language.

Nevertheless, it is important to keep in mind that programming language features, rather than Scala per se, are the focus of this book. Although I personally like it as a teaching language, I am not selling Scala, and this is not a Scala book. It just happens that I need a programming language that is clean and simple in all areas of interest, and I believe Scala meets these requirements.

## Target Audience

The target audience is programmers with enough experience to not be distracted by simple matters of syntax. I assume prior Java experience, or enough overall programming experience to read and understand simple Java code. Concepts such as classes, methods, objects, types, variables, loops, and conditionals are assumed to be familiar. A rudimentary understanding of program execution—execution stack, garbage collection, exceptions—is also assumed, as well as basic exposure to data structures and algorithms. For other key terms covered in depth in the book, the glossary provides a basic definition and indicates the appropriate chapter or chapters where the concept is presented.

*No prior knowledge of functional or concurrent programming is assumed. No prior knowledge of Scala is assumed.* Presumably, many readers will have some understanding of functional or concurrent concepts, such as recursion or locks, but no such knowledge is required. For instance, I do not expect you to necessarily understand the functional Python and Java programs discussed earlier, or the two Java concurrent programs, or the last two Scala functions. Indeed, I would argue that if these programs feel strange and mysterious, this book is for you! By comparison, the imperative variant of the number-shifting program should be easy to follow, and I expect you to understand the corresponding code, whether it is written in Java, Python, or Scala. You are expected to understand simple Scala syntax when it is similar to that of other languages and to pick up new elements as they are introduced.

The syntax of Scala was inspired by Java's syntax—and that of Java by C's syntax—which should make the transition fairly straightforward for most programmers. Scala departs from Java in ways that will be explained as code examples are introduced. For now, I'll highlight just three differences:

- *Semicolon inference.* In Scala, terminating semicolons are inferred by the compiler and rarely used explicitly. They may still appear occasionally—for instance, as a way to place two statements on the same line.

- *No "return" needed.* Although a `return` keyword exists in Scala, it is seldom used. Instead, a function implicitly returns the value of the last expression evaluated in its body.
- *Significant indentation.* The curly braces used to define code blocks can often be inferred from indentation and are optional. The first Scala `randShift` variant can been written:

  Scala
  ```scala
  def randShift(nums: List[Int], rand: Random): List[Int] =
     val shiftedNums = List.newBuilder[Int]
     for num <- nums do
        val shifted = num + rand.between(-10, 11)
        if shifted > 0 then shiftedNums += shifted
     end for
     shiftedNums.result()
  end randShift
  ```

  When indentation is used to create blocks, markers can be added to emphasize block endings, but they are optional. An even shorter version of the `randShift` function takes the following form:

  Scala
  ```scala
  def randShift(nums: List[Int], rand: Random): List[Int] =
     val shiftedNums = List.newBuilder[Int]
     for num <- nums do
        val shifted = num + rand.between(-10, 11)
        if shifted > 0 then shiftedNums += shifted
     shiftedNums.result()
  ```

  In this book, code illustrations rely on indentation instead of curly braces when possible and omit most end markers for the sake of compactness. I expect readers to be able to read imperative Scala code in this form, like the preceding function.

## How to Read This Book

I believe that the primary value of this book lies in its code illustrations. To a large extent, the text is there to support the code, more than the other way around. The code examples tend to be short and focused on the concepts they aim to illustrate. In particular, very few examples are designed to perform the specific tasks you need to solve in your daily programming activities. This is not a cookbook.

Furthermore, concepts are introduced from the ground up, starting with the fundamentals, and expanding and abstracting toward the application level. The code that you might find to be the most applicable is found in the later chapters in each part of the book. I have found this progression to be most conducive to a solid understanding of features, which can then be translated into languages other than Scala. If you feel that the early topics are well known and the pace too slow, please be patient.

This book is designed to be read in order, from beginning to end. Most chapters—and their code illustrations—depend on ideas and programs presented in earlier chapters. For instance, several solutions to the same problem are often presented in separate chapters as a way to illustrate different sets of programming language features. It is also the case that Part II on concurrent programming uses concepts from Part I on functional programming.

While this makes it near impossible to proceed through the contents in a different order, you are free to speed through sections that cover features with which you are already familiar. Material from this book has been to used to teach undergraduate and graduate students who are told that, as long as the code makes sense, they are ready to move on to the next part. It is when code starts to look puzzling that it is time to slow down and pay closer attention to the explanations in the text.

There are several ways you can safely skip certain parts of the contents:

- Chapter 15 on types can be skipped entirely. Elsewhere in the book, several code examples make simplifying assumptions to avoid intricate concepts such as type bounds and type variance. A basic understanding of Java types, including generics (but not necessarily with wildcards) and polymorphism, is sufficient.

- Any "aside" can be safely ignored. These are designed as complementary discussions that you may expect to find, given the book's topics (and I would not want to disappoint you!), and they can sometimes be lengthy. They are rarely referred to in the main text, and any of these references can be ignored.

- Any "case study" chapter can be skipped. I would not necessarily recommend that you do so, however, because the case study code is where features are put together in the most interesting ways. However, no concept or syntax needed in a later part of the book is ever introduced in a case study. The main text does not refer to code from the case studies, with one minor exception: Section 10.8 refers to a binary search tree implementation developed in Chapter 8.

## Additional Resources

The book's companion website is hosted at https://fcpbook.org. It contains additional resources, a list of errata, and access to the code illustrations, which are available from GitHub. The code examples were compiled and tested using Scala 3.2. The author welcomes comments and discussions, and can be reached at author@fcbbook.org.

# Acknowledgments

I want to thank past and present colleagues for the encouragements that got me started and for their feedback on the early stages of this project. Some were confident I had something to say (and could say it) before I realized it myself.

A special thanks to my students, who went through various iterations of the material that ended up in this book. They were my guinea pigs. More times than I care to admit, I subjected them to frantic improvisation because a feature suddenly needed for that day's lecture was only going to be introduced as part of the following week's discussion. (To arrange hundreds of code examples in a consistent order is harder than it looks.) With last minute changes before every class, students got used to, in their own words, "handouts and slides on which the ink is not quite yet dry."

Some of the ideas for code illustrations in this book were gathered over a period of thirty years, during which time I refined them by writing and rewriting many implementations in a variety of programming languages. As much as I'd like to specifically thank the original authors of these examples, I can't remember all the sources I've used, I don't know which of them were original, and I feel it wouldn't be fair to mention some names but not the others. Nevertheless, I don't claim to have invented all the examples used in this book. The code is mine (including bugs), but credit for program ideas that originated elsewhere should go to their creators, whoever they are.

It is truly scary to think what this book would have been if I had been left on my own. Whatever its current flaws, it was made astronomically better with the help of my editor, Gregory Doench, and his production team. They were very patient with a first-time author who clearly didn't always know what he was doing.

My feelings toward anonymous and non-anonymous reviewers is mixed. Without a doubt, they helped improve the book but at the cost of extending my prison sentence every time I was hoping to get paroled. I am thankful for their help—feedback from Cay Horstmann, Jeff Langr, and Philippe Quéinnec, and long email discussions with Brian Goetz, in particular, come to mind—but I cannot say I always welcomed their input as unmitigated good news. I had been warned that writing a book like this was a major undertaking but not that I would have to write it four times.

Which brings me to my deepest gratitude. It goes to my family, who showed angelic patience as I kept promising to be done "by next month" for more than a year. They must have grown tired of "the book" repeatedly getting in the way of our family life. Indeed, I'm amazed that my wife didn't pick up the phone one day, call my editor, and notify him: "That's it. The book is finished. Done. Today. Now."

*This page intentionally left blank*

# About the Author

**Michel Charpentier** is an associate professor with the Computer Science department at the University of New Hampshire (UNH). His interests over the years have ranged from distributed systems to formal verification and mobile sensor networks. He has been with UNH since 1999 and currently teaches courses in programming languages, concurrency, formal verification, and model-checking.

*This page intentionally left blank*

# Chapter 9

# Higher-Order Functions

It is natural for a programming paradigm that centers on functions to treat them as first-class citizens. In functional programming, functions are values and can be stored in variables and passed as arguments. Functions that consume or produce other functions are said to be higher-order functions. Using higher-order functions, computations can be parameterized by other computations in powerful ways.

## 9.1 Functions as Values

Previous chapters have shown how pure functions from immutable values to immutable values can be used as building programming blocks, and how complex computations can be achieved by composing functions, including composing a function with itself through recursion. Although pure functions, immutability, and recursion are essential concepts, many would argue that the distinctive characteristic of a functional programming style is the use of functions as values.

To help motivate the benefits of functions as values, consider this first illustration. Suppose you need to search for a value in a list. You can implement such a lookup in a recursive function, similar to function `contains` from Chapter 7:

*Scala*

```scala
def find[A](list: List[A], target: A): Option[A] = list match
   case Nil    => None
   case h :: t => if h == target then Some(h) else find(t, target)
```

Listing 9.1: List lookup for a specific target.

This function checks whether the head of a non-empty list equals the target and, if not, keeps searching in the tail of the list. It returns an option to allow for cases where the target value is not found. You can use `find` to look for specific values in a list, like a list of temperatures:

*Scala*

```scala
val temps = List(88, 91, 78, 69, 100, 98, 70)
find(temps, 78) // Some(78)
find(temps, 79) // None
```

A limitation of this function, however, is that you can only search for a target if you already have a value equal to that target. For instance, you cannot search a list of temperatures for a value greater than 90. Of course, you can easily write another function for that:

*Scala*
```
def findGreaterThan90(list: List[Int]): Option[Int] = list match
   case Nil    => None
   case h :: t => if h > 90 then Some(h) else findGreaterThan90(t)

findGreaterThan90(temps) // Some(91)
```

But what if you need to search for a temperature greater than 80 instead? You can write another function, in which an integer argument replaces the hardcoded value 90:

*Scala*
```
def findGreaterThan(list: List[Int], bound: Int): Option[Int] = list match
   case Nil    => None
   case h :: t => if h > bound then Some(h) else findGreaterThan(t, bound)

findGreaterThan(temps, 80) // Some(88)
```

This is better, but the new function still cannot be used to search for a temperature *less* than 90, or for a string that ends with `"a"`, or for a project with identity 12345.

You will notice that functions `find`, `findGreaterThan90`, and `findGreaterThan` are strikingly similar. The algorithm is the same in all three cases. The only part of the implementation that changes is the test in the `if-then-else`, which is `h == target` in the first function, `h > 90` in the next, and `h > bound` in the third.

It would be nice to write a generic function `find` parameterized by a search criterion. Criteria such as “to be greater than 90” or “to end with `"a"`” or “to have identity 12345” could then be used as arguments. To implement the `if-then-else` part of this function, you would apply the search criterion to the head of the list to produce a Boolean value. In other words, you need the search criterion to be a *function* from `A` to `Boolean`.

Such a function `find` can be written. It takes another function as an argument, named `test`:

*Scala*
```
def find[A](list: List[A], test: A => Boolean): Option[A] = list match
   case Nil    => None
   case h :: t => if test(h) then Some(h) else find(t, test)
```
Listing 9.2: Recursive implementation of higher-order function `find`.

The type of argument `test` is `A => Boolean`, which in Scala denotes functions from `A` to `Boolean`. As a function, `test` is applied to the head of the list `h` (of type `A`), and produces a value of type `Boolean` (used as the `if` condition).

You can use this new function `find` to search a list of temperatures for a value greater than 90 by first defining the "greater than 90" search criterion as a function:

Scala
```
def greaterThan90(x: Int): Boolean = x > 90
find(temps, greaterThan90) // Some(91)
```

In this last expression, you do not invoke function `greaterThan90` on an integer argument. Instead, you use the function itself as an argument to `find`. To search for a project with identity 12345, simply define a different search criterion:

Scala
```
def hasID12345(project: Project): Boolean = project.id == 12345L
find(projects, hasID12345) // project with identity 12345
```

Because it takes a function as an argument, `find` is said to be a *higher-order* function. Functional programming libraries define many standard higher-order functions, some of which are discussed in Chapter 10. In particular, a method `find` is already defined on Scala's `List` type. The two searches in the preceding examples can be written as follows:

Scala
```
temps.find(greaterThan90)
projects.find(hasID12345)
```

From now on, code examples in this chapter use the standard method `find` instead of the earlier user-defined function.

Method `find` is a higher-order function because it takes another function as an argument. A function can also be higher-order by returning a value that is a function. For example, instead of implementing `greaterThan90`, you can define a function that builds a search criterion to look for temperatures greater than a given bound:

Scala
```
def greaterThan(bound: Int): Int => Boolean =
   def greaterThanBound(x: Int): Boolean = x > bound
   greaterThanBound
```

Listing 9.3: Example of a function that returns a function; see also Lis. 9.4 and 9.5.

Function `greaterThan` works by first defining a function `greaterThanBound`. This function is not applied to anything but simply returned as a value. Note that `greaterThan` has return type `Int => Boolean`, which denotes functions from integers to Booleans. Given a lower bound `b`, the expression `greaterThan(b)` is a function, which tests whether an integer is greater than `b`. It can be used as an argument to higher-order method `find`:

*Scala*
```scala
temps.find(greaterThan(90))
temps.find(greaterThan(80))
```

In a similar fashion, you can define a function to generate search criteria for projects:

*Scala*
```scala
def hasID(identity: Long): Project => Boolean =
   def hasGivenID(project: Project): Boolean = project.id == identity
   hasGivenID

projects.find(hasID(12345L))
projects.find(hasID(54321L))
```

## 9.2 Currying

Functions that return other functions are common in functional programming, and many languages define a more convenient syntax for them:

*Scala*
```scala
def greaterThan(bound: Int)(x: Int): Boolean = x > bound
def hasID(identity: Long)(project: Project): Boolean = project.id == identity
```

Listing 9.4: Example of higher-order functions defined through currying.

It might appear as if `greaterThan` is a function of two arguments, `bound` and `x`, but it is not. It is a function of a single argument, `bound`, which returns a function of type `Int => Boolean`, as before; `x` is actually an argument of the function being returned.

Functions written in this style are said to be *curried*.[1] A curried function is a function that consumes its first list of arguments, returns another function that uses the next argument list, and so on. You can read the definition of `greaterThan` as implementing a function that takes an integer argument `bound` and returns another function, which takes an integer argument `x` and returns the Boolean `x > bound`. In other words, the return value of `greaterThan` is the function that maps `x` to `x > bound`.

Functional programming languages rely heavily on currying. In particular, currying can be used as a device to implement all functions as single-argument functions, as in

[1]The concept is named after the logician Haskell Curry, and the words *curried* and *currying* are sometimes capitalized.

languages like Haskell and ML. For instance, we tend to think of addition as a function of two arguments:

```scala
def plus(x: Int, y: Int): Int = x + y // a function of type (Int, Int) => Int
plus(5, 3)                            // 8
```

However, you can also think of it as a single-argument (higher-order) function:

```scala
def plus(x: Int)(y: Int): Int = x + y // a function of type Int => (Int => Int)
plus(5)                               // a function of type Int => Int
plus(5)(3)                            // 8
```

Curried functions are so common in functional programming that the `=>` that represents function types is typically assumed to be right-associative: `Int => (Int => Int)` is simply written `Int => Int => Int`. For example, the function

```scala
def lengthBetween(low: Int)(high: Int)(str: String): Boolean =
   str.length >= low && str.length <= high
```

has type `Int => Int => String => Boolean`. You can use it to produce a Boolean, as in

```scala
lengthBetween(1)(5)("foo") // true
```

but also to produce other functions:

```scala
val lengthBetween1AndBound: Int => String => Boolean = lengthBetween(1)
val lengthBetween1and5: String => Boolean            = lengthBetween(1)(5)

lengthBetween1AndBound(5)("foo") // true
lengthBetween1and5("foo")        // true
```

Before closing this section on currying, we should consider a feature that is particular to Scala (although other languages use slightly different tricks for the same purpose).

In Scala, you can call a single-argument function on an expression delimited by curly braces without the need for additional parentheses. So, instead of writing

*Scala*
```
println({
   val two = 2
   two + two
}) // prints 4
```

you can simply write:

*Scala*
```
println {
   val two = 2
   two + two
} // prints 4
```

To use this syntax when multiple arguments are involved, you can rely on currying to adapt a multi-argument function into a single-argument function. For instance, the curried variant of function `plus` can be invoked as follows:

*Scala*
```
plus(5) {
   val two = 2
   two + 1
}
```

This is still value 8, as before.

Many functions and methods are curried in Scala for the sole purpose of benefiting from this syntax. The syntax is introduced here because we will encounter some example uses throughout the book, starting with the next section.

## 9.3 Function Literals

It would be inconvenient if, to use higher-order functions like `find`, you always had to separately define (and name) argument functions like `hasID12345` and `greaterThan90`. After all, when you call a function on a string or an integer, you don't need to define (and name) the values first. This is because programming languages define a syntax for strings and integer literals, like the `"foo"` and `42` that are sprinkled throughout this book's code illustrations. Similarly, functional programming languages, which rely heavily on higher-order functions, offer syntax for *function literals*, also called *anonymous functions*. The most common form of function literals is lambda expressions, which are often the first thing that comes to mind when you hear that a language has support for functional programming.

In Scala, the syntax for lambda expressions is `(v1: T1, v2: T2, ...) => expr`.[2] This defines a function with arguments `v1`, `v2`, ... that returns the value produced by `expr`. For instance, the following expression is a function, of type `Int => Int`, that adds 1 to an integer:

Scala
```scala
(x: Int) => x + 1
```

Function literals can be used to simplify calls to higher-order functions like `find`:

Scala
```scala
temps.find((temp: Int) => temp > 90)
projects.find((proj: Project) => proj.id == 12345L)
```

The Boolean functions "to be greater than 90" and "to have identity 12345" are implemented as lambda expressions, which are passed directly as arguments to method `find`.

You can also use function literals as return values of other functions. So, a third way to define functions `greaterThan` and `hasID`, besides using named local functions or currying, is as follows:

Scala
```scala
def greaterThan(bound: Int): Int => Boolean = (x: Int) => x > bound
def hasID(identity: Long): Project => Boolean = (p: Project) => p.id == identity
```

Listing 9.5: Example of higher-order functions defined using lambda expressions.

The expression `(x: Int) => x > bound` replaces the local function `greaterThanBound` from Listing 9.3.

Function literals have no name, and usually do not declare their return type. Compilers can sometimes infer the types of their arguments. You could omit argument types in all the examples written so far:

Scala
```scala
temps.find(temp => temp > 90)
projects.find(proj => proj.id == 12345L)

def greaterThan(bound: Int): Int => Boolean = x => x > bound
def hasID(identity: Long): Project => Boolean = p => p.id == identity
```

[2]Lambda expressions can also be parameterized by types, though this is a more advanced feature not used in this book. For instance, Listing 2.7 defines a function `first` of type `(A, A) => A`, parameterized by type `A`. It could be written as the lambda expression `[A] => (p: (A, A)) => p(0)`.

Today, many programming languages have a syntax for function literals. The Scala expression `(temp: Int) => temp > 90` could be written in other languages as shown here:

```
(int temp) -> temp > 90                 // Java
(int temp) => temp > 90                 // C#
[](int temp) { return temp > 90; }      // C++
{ temp: Int -> temp > 90 }              // Kotlin
fn temp: int => temp > 90               // ML
fn temp -> temp > 90 end                // Elixir
function (temp) { return temp > 90 }    // JavaScript
temp => temp > 90                       // also JavaScript
lambda { |temp| temp > 90 }             // Ruby
-> temp { temp > 90 }                   // also Ruby
(lambda (temp) (> temp 90))             // Lisp
(fn [temp] (> temp 90))                 // Clojure
lambda temp: temp > 90                  // Python
```

The argument (or arguments) of a lambda expression can be composite types. For example, assume you have a list of pairs (*date*, *temperature*), and you need to find a temperature greater than 90 in January, February, or March. You can use `find` with a lambda expression on pairs:

*Scala*
```scala
val datedTemps: List[(LocalDate, Int)] = ...
datedTemps.find(dt => dt(0).getMonthValue <= 3 && dt(1) > 90)
```

The test checks that the first element of a pair (a date) is in the first three months of the year, and that the second element of the pair (a temperature) is greater than 90.

Languages that support pattern matching often let you use it within a lambda expression. In the preceding example, you can use pattern matching to extract the date and temperature from a pair, instead of `dt(0)` and `dt(1)`:

*Scala*
```scala
datedTemps.find((date, temp) => date.getMonthValue <= 3 && temp > 90)
```

This is a lot more readable than the variant that uses `dt(0)` and `dt(1)`.

More complex patterns can be used. In Scala, a series of `case` patterns, enclosed in curly braces, also define an anonymous function. For instance, if a list contains temperatures with an optional date, and temperatures without a date are not eligible, you can search for a temperature greater than 90 in the first three months with the following code:[3]

[3]Here, I must admit that the Scala syntax can be confusing at first. As with code blocks, a call `f({...})` can omit extraneous parentheses, and be written as `f{...}`. This example becomes clearer once you understand that it is a call to a higher-order method on a function literal defined with pattern matching and that a pair of unnecessary parentheses have been dropped.

Scala
```scala
val optionalDatedTemps: List[(Option[LocalDate], Int)] = ...

optionalDatedTemps.find {
   case (Some(date), temp) => date.getMonthValue <= 3 && temp > 90
   case _                  => false
}
```

## 9.4 Functions Versus Methods

So far in this book, the words *function* and *method* have been used almost interchangeably. It is now time to discuss differences between the two. Methods are often defined as being functions associated with a target object: `x.m(y)`, which invokes method `m` on object `x` with argument `y`, is not much different from `f(x,y)`, which calls a function `f` on `x` and `y`. This way of differentiating methods from functions is premised on them being mechanisms used to encapsulate behaviors—blocks of code—which both methods and functions are.

However, the story somewhat changes once functions become values. A more meaningful difference between methods and functions in the context of this book is that functions are values in functional programming, while methods are not objects in object-oriented programming. In a hybrid language like Scala, functions *are* objects; methods are not. Instead of the function literal `(temp: Int) => temp > 90`, you could build a regular object explicitly. This object would implement the type `Function`:

Scala
```scala
object GreaterThan90 extends Function[Int, Boolean]:
   def apply(x: Int): Boolean = x > 90
```

The notation `Int => Boolean` is syntactic sugar for the type `Function[Int, Boolean]`. This type defines a method `apply`, which is invoked when the function is applied. The expression `temps.find(GreaterThan90)` could replace `temps.find(temp => temp > 90)` to perform the same computation.[4] `GreaterThan90` is an object—which defines a function—not a method. In contrast,

Scala
```scala
def greaterThan90(x: Int): Boolean = x > 90
```

defines a method `greaterThan90`, not a function.

But then, the plot thickens. We *did* write `temps.find(greaterThan90)` earlier to search a list of temperatures, as if `greaterThan90` were an object, which it is not. This is possible because the language implements bridges between methods and functions. In

[4]In JVM languages, anonymous functions are often compiled through a separate mechanism but, as a function, object `GreaterThan90` is conceptually equivalent to the earlier lambda expression.

Section 2.5, we discussed extension methods, a mechanism to make a function appear as a method. Here, what we need is a conversion in the opposite direction so that we can use a method as a function.

The fancy name for this is *$\eta$-conversion*. In $\lambda$-calculus, it states the equivalence between $f$ and $\lambda x.\, f\, x$. In plainer terms, you can think of it as an equivalence between `greaterThan90` and `x => greaterThan90(x)`. As units of computation, both perform the same task of asserting whether an integer is greater than 90. Given that the argument of `find` must have type `Int => Boolean`, and `greaterThan90` is a method from `Int` to `Boolean`, the intent of the expression `temps.find(greaterThan90)` is pretty clear, and the compiler is able to insert the necessary $\eta$-conversion.

Other languages offer similar bridges to create a function out of a method, sometimes by relying on a more explicit syntax. In Java, for instance, a lambda expression `x -> target.method(x)` can be replaced with a method reference `target::method`. Kotlin uses a similar syntax.

## 9.5 Single-Abstract-Method Interfaces

In hybrid languages, functions are objects, and lambda expressions are used as a convenient way to create such objects. Indeed, the lambda expression syntax is so handy that many languages let you use it to create instances of types other than functions.

A single-abstract-method (SAM) interface is an interface that contains exactly one abstract method. In Scala, for instance, the type `Function[A,B]` (or, equivalently, `A => B`) is a SAM interface with a single abstract method `apply`. We have used lambda expressions in code illustrations to create instances of `Function[A,B]`, but it turns out that all SAM types can be instantiated using lambda expressions, even types that are not related to `Function`:

*Scala*
```scala
abstract class Formatter:
   def format(str: String): String
   def println(any: Any): Unit = Predef.println(format(any.toString))
```

Class `Formatter` defines only one abstract method `format` and is therefore a SAM interface. It can be implemented using lambda expressions:

*Scala*
```scala
val f: Formatter = str => str.toUpperCase
f.println(someValue)
```

Note how method `println` is called on object `f`, which was defined as a lambda expression. This is possible only because `f` was declared with type `Formatter`; the expression `(str => str.toUpperCase).println("foo")` would make no sense.

Many Java interfaces can be implemented as lambdas, even though they predate Java's syntax for lambda expressions and have little to do with functional programming:

```scala
val absComp: Comparator[Int] = (x, y) => x.abs.compareTo(y.abs)

val stream: IntStream = ...
val loggingStream: IntStream = stream.onClose(() => logger.info("closing stream"))
```

`Comparator` is a Java 2 SAM interface with an abstract method `compare`. Stream method `onClose` uses a single argument of type `Runnable`, a Java 1 SAM interface with an abstract method `run`. Both `Comparator` and `Runnable` can be implemented as lambda expressions.

## 9.6 Partial Application

In addition to lambda expressions, currying, and $\eta$-conversion, partial application is yet another mechanism used to create function values. In Scala, it takes the form of an underscore used in place of a part of an expression. This produces a function that, when applied, replaces the underscore with its argument in the given expression. For instance, the following code searches for a temperature greater than 90 (in Fahrenheit) in a list of Celsius temperatures:

```scala
celsiusTemps.find(temp => temp * 1.8 + 32 > 90)
```

Instead of a lambda expression, you can build the desired function argument by replacing `temp` with an underscore in the expression `temp * 1.8 + 32 > 90`:

```scala
celsiusTemps.find(_ * 1.8 + 32 > 90)
```

The expression `_ * 1.8 + 32 > 90` represents a Boolean function that maps `temp` to `temp * 1.8 + 32 > 90`, just like the function defined by the lambda expression `temp => temp * 1.8 + 32 > 90`. Searches written earlier using lambda expressions can use partial application instead:

```scala
temps.find(_ > 90)
projects.find(_.id == 12345L)
```

Partial application is generalized to multi-argument functions by using several underscores in the same expression. For instance, `_ * 1.8 + 32 > _` is a two-argument function that compares a Celsius temperature to a Fahrenheit bound, and `_.id == _` is a function that takes a project and an identity and checks whether the project has the given identity.[5]

Partial application can easily be abused. Code is often easier to read with lambda expressions that name (and sometimes type) their arguments. Compared to the shorter `_.id == 12345L`, the longer expression `project => project.id == 12345L` makes it clearer that projects are being searched, and `(project, id) => project.id == id` is a lot easier to read than `_.id == _`.

**Aside on Scoping**

After a variable is introduced in a program, the variable's name is bound to that variable's value. The part of the program where this binding exists is called the *scope* of the declaration. Scoping rules vary from programming language to programming language. The state of affairs is somewhat simpler than it used to be because almost all languages rely on *static* (or *lexical*) scoping. Only a few languages continue to offer a form of *dynamic* (or *late binding*) scoping. However, many popular languages implement their static scoping rules differently, so caution is still warranted.

The following Scala program involves multiple scopes:

*Scala*
```scala
var str: String = ""

def f(x: Int): Int =
   if x > 0 then
      val str: Int = x - 1
      str + 1
   else
      val x: String = str.toUpperCase
      x.length
```

The outermost scope defines a variable `str`, of type `String`. The body of function `f` creates its own scope, in which a variable `x`, of type `Int`, is defined (the argument to the function). The block of code that constitutes the `then` part of the conditional has its own scope in which a new variable `str`, of type `Int`, is declared. Similarly, the `else` block defines a new variable `x`, of type `String`, in its own scope. Variables do not exist outside their scopes: Outside function `f`, variable `str` is the string defined on the first line, and there is no variable named `x`.

The variables `str` and `x` declared in the inner scopes *shadow* the variables with the same names from the outer scopes. Java forbids such shadowing and

[5] Be careful, because details vary from language to language. For instance, while "`_ + _`" is a two-argument adding function in Scala, `it + it` is a single-argument doubling function in Kotlin. In Elixir, `&(&1 + &2)` is the two-argument adding function, and `&(&1 + &1)` is the single-argument doubling function.

forces you to pick different names for the variables declared inside the `then` and `else` blocks.

While it is often the case that every block of code defines its own scope, not all languages adhere to this rule. JavaScript and Python, for instance, introduce a new scope for the body of a function, but *not* for the `then` and `else` branches of a conditional, or the body of a loop. This can be confusing when you are used to the more mainstream scoping rules. As an illustration, consider the following Python program:

*Python*

```python
x = 1

def f():
    x = 2
    if x > 0:
        x = 3
        y = 4
    print(x) # prints 3
    print(y) # prints 4

f()
print(x) # prints 1
print(y) # error: name 'y' is not defined
```

The body of function `f` defines its own scope, but the block of code inside `if` does not. Instead, `x=3` is an assignment to the variable `x` declared in the scope of the function (initialized with 2), and `y=4` introduces a variable `y` inside that same scope. In particular, this variable `y` continues to exist after the `if` statement and has value 4. The `print(x)` statement inside function `f` prints the value of the variable `x` in the scope of the function, which is 3, while the `print(x)` statement outside function `f` prints the value of the variable `x` in the outermost scope, which is 1. The final `print(y)` statement triggers an error, since no `y` variable has been declared outside the scope of the function. The behavior would be the same in JavaScript. Contrast this with Scala:

*Scala*

```scala
var x = 1

def f() =
   var x = 2
   if x > 0 then
      var x = 3
      var y = 4
   println(x) // prints 2
   println(y) // rejected at compile-time
```

```
f()
println(x) // prints 1
println(y) // rejected at compile-time
```

Most languages now use static scoping, with variations as to which programming language constructs introduce a new scope. Dynamic scoping, in contrast, is error-prone and has become less popular. It was used in the original Lisp and remains available as an option in modern variants of that language. It is also used in some scripting languages, most notably Perl and various Bourne Shell implementations.

As an illustration, this Scala program follows static scoping rules:

*Scala*

```
var x = 1

def f() =
   x += 1
   println(x) // prints 2

def g() =
   var x = 10
   f()

g()
println(x) // prints 2
```

Function `g` defines a local variable `x`, then invokes `f`. The variable `x` used inside function `f`, however, is the one declared on the first line of the program, which is the one in scope where function `f` is defined. The local variable with the same name defined inside function `g` plays no part. The behavior would be the same for an equivalent program written in Java, Kotlin, C, or any one of a multitude of languages that use static scoping.

Contrast this with the following Bash implementation:

*Bash*

```
x=1

f() {
  (( x++ ))
  printf "%d\n" $x # prints 11
}

g() {
  local x=10
  f
}

g
printf "%d\n" $x # prints 1
```

In Bash, the variable `x` used inside function `f` is not the variable in scope where `f` is defined, but the variable in scope where function `f` is invoked. This variable, equal to 10, is incremented to 11. The variable `x` declared at the beginning of the program was never modified.

Dynamic scoping can be used to override a global variable with a local variable, thus changing the behavior of a function. This is sometimes useful, and a similar behavior can be achieved in Scala through implicit arguments. For instance, by reusing the `Formatter` type from Section 9.5, a function can be defined to print an object with the default formatter in scope:

*Scala*
```scala
def printFormatted(any: Any)(using formatter: Formatter): Unit =
   formatter.println(any)

printFormatted("foo") // uses the default formatter in scope (there must be one)
```

Within a function—or any block of code that introduces its own scope—a different formatter can be specified:

*Scala*
```scala
given UpperCaseFormatter: Formatter = str => str.toUpperCase

printFormatted("foo") // prints "FOO"
```

This technique brings back some of the flexibility of dynamic scoping but is much safer: Function `printFormatted` is explicit in the fact that it allows a locally defined formatter to impact its behavior.

The code examples in this book do not rely much on implicit arguments, except on occasion in Part II. In particular, Scala tends to use implicit arguments to specify the thread pool on which to execute concurrent activities.

## 9.7 Closures

Recall our first implementation of function `greaterThan`, in Listing 9.3:

*Scala*
```scala
def greaterThan(bound: Int): Int => Boolean =
   def greaterThanBound(x: Int): Boolean = x > bound
   greaterThanBound
```

You can apply `greaterThan` to different values to produce different functions. For example, `greaterThan(5)` is a function that tests if a number is greater than 5, while `greaterThan(100)` is a function that tests if a number is greater than 100:

*Scala*
```scala
val gt5   = greaterThan(5)
val gt100 = greaterThan(100)

gt5(90)   // true
gt100(90) // false
```

The question to ponder is this: In the `gt5(90)` computation, which compares 90 to 5, where does the value 5 come from? A 5 was pushed on the execution stack as local variable `bound` for the call `greaterThan(5)`, but this call has been completed, and the value removed from the stack. In fact, another call has already taken place with local variable `bound` equal to 100. Still, `gt5(90)` compares to 5, not to 100. Somehow, the value 5 of variable `bound` was captured during the call `greaterThan(5)`, and is now stored as part of function `gt5`.

The terminology surrounding this phenomenon is somewhat ambiguous, but most sources define *closures* to be functions associated with captured data.[6] When a function, like `greaterThanBound`, uses variables in its body other than its arguments—here, `bound`—these variables must be captured to create a function value.

Closures are sometimes used in functional programming languages as a way to add state to a function. For instance, a function can be "memoized" (a form of caching) by storing the inputs and outputs of previous computations:

*Scala*
```scala
def memo[A, B](f: A => B): A => B =
   val store = mutable.Map.empty[A, B]

   def g(x: A): B =
      store.get(x) match
         case Some(y) => y
```

[6]The term *closure* is sometimes used to refer to the data only, or to the capturing phenomenon itself. The word *closure* comes from the fact that, in $\lambda$-calculus, a function like `greaterThanBound` is represented by an *open* term that contains a free variable `bound`, and that needs to be *closed* to represent an actual function.

```scala
         case None =>
            val y = f(x)
            store(x) = y
            y

   g
```

Listing 9.6: Memoization using closures; see also Lis. 12.2.

Function `memo` is a higher-order function. Its argument `f` is a function of type `A => B`. Its output is another function, `g`, of the same type. Function `g` is functionally equivalent to `f`—it computes the same thing—but stores every computed value into a map. When called on some input `x`, function `g` first looks up the map to see if value `f(x)` has already been calculated and if so, returns it. Otherwise, `f(x)` is computed, using function `f`, and stored in the map before being returned. You apply `memo` to a function to produce a memoized version of that function:

```scala
val memoLength: String => Int = memo(str => str.length)

memoLength("foo") // invokes "foo".length and returns 3
memoLength("foo") // returns 3, without invoking method length
```

Function `memoLength` is a function from strings to integers, like `str => str.length`. It calculates the length of a string and stores it. The first time you call `memoLength("foo")`, the function invokes method `length` on string `"foo"`, stores 3, and returns 3. If you call `memoLength("foo")` again, value 3 is returned directly, without invoking method `length` of strings. Another invocation `memo(str => str.length)` would create a new closure with its own `store` map.

What is captured by a closure is a lexical environment. This environment contains function arguments, local variables, and fields of an enclosing class, if any:

```scala
def logging[A, B](name: String)(f: A => B): A => B =
   var count  = 0
   val logger = Logger.getLogger("my.package")

   def g(x: A): B =
      count += 1
      logger.info(s"calling $name ($count) with $x")
      val y = f(x)
      logger.info(s"$name($x)=$y")
      y

   g
```

Listing 9.7: Example of a function writing in its closure.

Like `memo`, function `logging` takes a function of type `A => B` as its argument and produces another function of the same type. The returned function is functionally equivalent to the input function, but it adds logging information, including the input and output of each call and the number of invocations:

*Scala*
```scala
val lenLog: String => Int = logging("length")(str => str.length)

lenLog("foo")
// INFO: calling length (1) with foo
// INFO: length(foo)=3

lenLog("bar")
// INFO: calling length (2) with bar
// INFO: length(bar)=3
```

For this to work, the returned closure `g` needs to maintain references to arguments `name` and `f`, as well as to local variables `count` and `logger`.

Note that variable `count` is modified when the closure is called. Writing into closures can be a powerful mechanism, but it is also fraught with risks:

*Scala*
```scala
// DON'T DO THIS!
val multipliers = Array.ofDim[Int => Int](10)

var n = 0
while n < 10 do
   multipliers(n) = x => x * n
   n += 1
```

This code attempts to create an array of multiplying functions: It fills the array with functions of type `Int => Int` defined as `x => x * n`. The idea is that `multipliers(i)` should then be `x => x * i`, a function that multiplies its argument by `i`. However, as written, the implementation does not work:

*Scala*
```scala
val m3 = multipliers(3)
m3(100) // 1000, not 300
```

All the functions stored in the array close over variable `n` *and share it*. Since `n` is equal to 10 at the end of the loop, all the functions in the array multiply their argument by 10 (at least, until `n` is modified). Some languages, including Java, emphasize safety over flexibility and do not allow local variables captured in closures to be written.

As with other forms of implicit references (e.g., inner classes), you need to be aware of closures to avoid tricky bugs caused by unintended sharing. This is especially true when closing over mutable data. As always, emphasizing immutability tends to result in safer code.

## 9.8 Inversion of Control

You may sometimes see higher-order functions being discussed within the broader notion of inversion of control. When using higher-order functions, control flow moves from the caller into the callee, which uses function arguments as callbacks into the caller.

To search a list of temperatures for a value greater than 90, you can use the recursive function `findGreaterThan90` defined earlier in this chapter, or the loop-based equivalent:

*Scala*
```scala
def findGreaterThan90(list: List[Int]): Option[Int] =
   var rem = list
   while rem.nonEmpty do
      if rem.head > 90 then return Some(rem.head) else rem = rem.tail
   None
```

Whether you use recursion or a loop, the list is queried for its values (head and tail), but the flow of control remains within function `findGreaterThan90`. If instead you use the expression `temps.find(greaterThan90)`, you no longer query the list for its values. Function `find` is now responsible for the flow of control—and may use recursion or a loop, depending on its own implementation. It makes callbacks to your code, namely the test function `greaterThan90`.

This shift of control flow from application code into library code is one of the reasons functional programming feels more abstract and declarative compared to imperative programming. However, once higher-order functions are well understood, they become convenient abstractions that can improve productivity and reduce the need for debugging. By using a method like `find`, you not only save the time it takes to write the three or four lines needed to implement the loop, but more importantly, eliminate the risk of getting it wrong.

## 9.9 Summary

- A defining characteristic of functional programming is the use of functions as values. Functions can be stored in data structures, passed as arguments to other functions, or returned as values by other functions. Functions that take functions as arguments or return functions as values are said to be higher-order functions.

- Passing functions as arguments to other functions makes it possible to parameterize a higher-order function by the behavior of its function arguments—for example, a searching method parameterized by a search criterion.
- To facilitate the implementation and usage of higher-order functions, many programming languages offer a syntax for function literals, which are expressions that evaluate to function values. This can take different forms, but a very common syntax is that of a lambda expression, which defines an anonymous function in terms of its arguments and return value.
- A curried function consumes its first argument (or argument list) and returns another function that will use the remaining arguments (or argument lists). By currying, a function that uses a list of multiple arguments can be transformed into a function that uses multiple lists of fewer arguments. This facilitates partial application to some but not all of the original arguments.
- In hybrid languages that combine object-oriented and functional programming, function values tend to appear as objects, and a function is applied by invoking a method of the object. Functions and methods are thus conceptually different: Functions are objects, which contain methods. Note that both methods and functions can be higher-order.
- Hybrid languages define syntax to bridge methods and functions, specifically to build a function value out of code defined in a method. One example is method reference: `obj::method` represents the function `x -> obj.method(x)` in Java. Another is implicit $\eta$-conversion: `obj.method` is transformed by the Scala compiler into `x => obj.method(x)` based on context.
- Partial application is another convenience mechanism used to generate functions. It relies on placeholders—for example, "`_`" in Scala, `it` in Kotlin—that make a function out of an arbitrary expression and can be thought of as a generalization of currying.
- The syntax used to implement function literals—lambda expressions, method references, partial application, etc.—can often be used in hybrid languages to create instances of SAM interfaces, which are interfaces and abstract classes with a single abstract method. This results in frequent use of lambda expressions as a replacement for more verbose mechanisms, such as anonymous classes, independently from functional programming patterns.
- When a function value is produced from code in a function or method that refers to variables other than its arguments, the compiler needs to construct a closure. The closure associates the function being created with a lexical environment that captures those variables. This is necessary for a function value to be usable outside its defining context.

- Programming with higher-order functions often involves a form of inversion of control. Control flow is embedded into a higher-order function, which then uses arguments as callbacks into the caller's code. It is an effective programming style once mastered, but the resulting code is more abstract and can require some adjustment for programmers used to imperative programming.

*This page intentionally left blank*

# Chapter 26

# Functional-Concurrent Programming

Functional tasks can be handled as futures, which implement the basic synchronization needed to wait for completion and to retrieve computed values (or exceptions). As synchronizers, however, futures suffer from the same drawbacks as other blocking operations, including performance costs and the risk of deadlocks. As an alternative, futures are often enriched with higher-order methods that process their values asynchronously, without blocking. Actions with side effects can be registered as callbacks, but the full power of this approach comes from applying functional transformations to futures to produce new futures, a coding style this book refers to as *functional-concurrent programming*.

## 26.1 Correctness and Performance Issues with Blocking

In Chapter 25, we looked at futures as data-carrying synchronizers. Futures can be used to simplify concurrent programming, especially compared to error-prone, do-it-yourself, lock-based strategies (e.g., Listings 22.3 and 22.4).

However, because they block threads, all synchronizers suffer from the same weaknesses—and that includes futures. Earlier, we discussed the possibility of misusing synchronizers in such a way that threads end up waiting for each other in a cycle, resulting in a deadlock. Synchronization deadlocks can happen with futures as well and can actually be quite sneaky—running the server of Listing 25.2 on a single thread pool is a mistake that is easy to make.

Before going back to this server example, consider first, as an illustration, a naive implementation of a parallel quick-sort:

*Scala*

```scala
// DON'T DO THIS!
def quickSort(list: List[Int], exec: ExecutorService): List[Int] =
   list match
      case Nil => list
      case pivot :: others =>
         val (low, high) = others.partition(_ < pivot)
         val lowFuture   = exec.submit(() => quickSort(low, exec))
      val highSorted  = quickSort(high, exec)
      lowFuture.get() ::: pivot :: highSorted
```

Listing 26.1: Deadlock-prone parallel quick-sort; see also Lis. 26.5.

This method follows the same pattern as the previous implementation of quick-sort in Listing 10.1. The only difference is that the function uses a separate thread to sort the low values, while the current thread sorts the high values, thus sorting both lists in parallel. After both lists have been sorted, the sorted low values are retrieved using method `get`, and the two lists are concatenated around the pivot as before. This is the same pattern used in the server example to fetch a customized ad in the background except that the task used to create the future is the function itself, recursively.

At first, the code appears to be working well enough. You can use `quickSort` to successfully sort a small list of numbers:

*Scala*
```scala
val exec = Executors.newFixedThreadPool(3)
quickSort(List(1, 6, 8, 6, 1, 8, 2, 8, 9), exec) // List(1, 1, 2, 6, 6, 8, 8, 8, 9)
```

However, if you use the same three-thread pool in an attempt to sort the list `[5,4,1,3,2]`, the function gets stuck and fails to terminate. Looking at a thread dump would show that all three threads are blocked on a call `lowFuture.get` in a deadlock.

Figure 26.1 displays the state of the computation at this point as a tree. Each sorting task is split into three branches: `low`, `pivot`, and `high`. The thread that first invokes `quickSort` is called `main` here. It splits the list into a `pivot` (5), a `low` list (`[4,1,3,2]`), and a `high` list (`[]`). It quickly sorts the empty list itself, and then blocks, waiting for the sorting of list `[4,1,3,2]` to complete. Similar steps are taking place with the thread

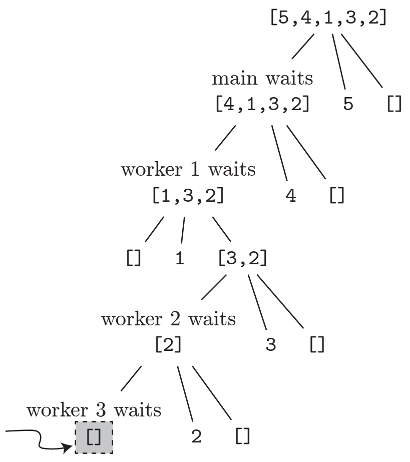

**Figure 26.1 Deadlock as a result of tasks created by tasks.**

in charge of sorting list [4,1,3,2], and so on, recursively. In the end, the three workers from the thread pool are blocked on three sorting tasks: [1,3,2], [2], and []. This last task—sorting the empty list at the bottom left of Figure 26.1—sits in the queue of the thread pool, as there is no thread left to run it.

This faulty implementation of quick-sort works adequately on computations with short `low` lists, even if the `high` lists are long. You can use it to sort the already sorted list [1,2,...,100] on a three-thread pool, for instance. However, the function fails when `low` lists are long, even if `high` lists are short. On the same three-thread pool, it cannot sort the list [5,4,1,3,2], or even the list [4,3,2,1].

Tasks that recursively create more tasks on the same thread pool are a common source of deadlocks. In fact, that problem is so prevalent that special thread pools were designed to handle it (see Section 27.3). Recursive tasks will get you in trouble easily, but as soon as tasks wait for completion of other tasks on the same thread pool, the risk of deadlock is present, even without recursion. This is why the server in Listing 25.2 uses two separate thread pools. Its `handleConnection` function—stripped here of code that is not relevant to the discussion—involves the following steps:

*Scala*

```scala
val futureAd: Future[Ad] = exec.submit(() => fetchAd(request)) // a Java future
val data: Data           = dbLookup(request)
val page: Page           = makePage(data, futureAd.get())
connection.write(page)
```

Listing 26.2: Ad-fetching example; contrast with Lis. 26.3, 26.6, and 26.7.

With a single pool of $N$ threads, you could end up with $N$ simultaneous connections, and thus $N$ concurrent runs of function `handleConnection`. Each run would submit an ad-fetching task to the pool, with no thread to execute it. All the runs would then be stuck, forever waiting on `futureAd.get`.

Avoiding these deadlock situations is typically not easy. You may have to add many threads to a pool to make sure that deadlocks cannot happen, but large numbers of threads can be detrimental to performance. It is quite possible—likely, even—that most runs will block only a small subset of threads, nowhere near a deadlock situation, and leave too many active threads that use CPU resources. Some situations are hopeless: In the worst case, the naive quick-sort example would need as many threads as there are values in the list to guarantee that a computation remains free of deadlock.

Even if you find yourself in a better situation and deadlocks can be avoided with a pool of moderate size, waiting on futures still incurs a non-negligible cost. Blocking—on any kind of synchronizer, including locks—requires parking a thread, saving its execution stack, and later restoring the stack and restarting the thread. A parked thread also tends to see its data in a processor-level cache overwritten by other computations, resulting in cache misses when the thread resumes execution. This can have drastic consequences on performance.

Avoiding deadlocks should, of course, be your primary concern, but these performance costs cannot always be ignored. Thus, they constitute another incentive to reduce thread blocking. Several strategies have been proposed to minimize blocking, and some are described in detail in Chapter 27. For now, we will focus on a functional-concurrent programming style that uses futures through higher-order functions without blocking. It departs from the more familiar reliance on synchronizers and as such takes some getting used to. Once mastered, it is a powerful way to arrange concurrent programs.

## 26.2 Callbacks

**NOTE**

The code illustrations in this chapter rely mostly on Scala's futures for the same reason the book uses Scala in the first place: They tend to be cleaner than (though not always as rich as) Java's `CompletableFuture`. Listings 26.10 and 27.9 and some of the code in Chapter 28 make use of `CompletableFuture`, with more examples in Appendix A. Note also that callback mechanisms and other higher-order functions on futures often require an execution context—typically a thread pool in concurrent applications. How this context is specified varies from language to language. It can also become a distraction when presenting more important concepts. Most functions in this chapter assume a global execution context, which is left unspecified. One exception is Listing 26.5 (for consistency with Listing 26.1); other functions could use a similar pattern—that is, add a "`(using ExecutionContext)`" argument instead of assuming a global context.

A callback is a piece of code that is registered for execution, often later (asynchronous callback). Modern implementations of futures—including Scala's `Future` and Java's `CompletableFuture`—offer a callback-registration mechanism. On a Scala future, you register a callback using method `onComplete`, which takes as its argument an action to apply to the result of the future. Because a future can end up with an exception instead of a value, the input of a callback action is of type `Try` (see Section 13.3).

On a future whose task is still ongoing, a call to `onComplete` returns immediately. The action will run when the future finishes, typically on a thread pool specified as execution context:

*Scala*
```scala
println("START")
given ExecutionContext = ... // a thread pool

val future1: Future[Int]    = ... // a future that succeeds with 42 after 1 second
val future2: Future[String] = ... // a future that fails with NPE after 2 seconds
future1.onComplete(println)
future2.onComplete(println)

println("END")
```

This example starts a 1-second task and a 2-second task and registers a simple callback on each. It produces an output of the following form:

```
main at XX:XX:33.413: START
main at XX:XX:33.465: END
pool-1-thread-3 at XX:XX:34.466: Success(42)
pool-1-thread-3 at XX:XX:35.465: Failure(java.lang.NullPointerException)
```

You can see that the main thread terminates immediately—callback registration takes almost no time. One second later, the first callback runs and prints a `Success` value. One second after that, the second callback runs and prints a `Failure` value. In this output, both callbacks ran on the same thread, but there is no guarantee that this will always be the case.

You can use a callback in the ad-fetching scenario. Instead of waiting for a customized ad to assemble a page, as in Listing 26.2, you specify as an action what is to be done with the ad once it becomes available:

*Scala*

```scala
val futureAd: Future[Ad] = Future(fetchAd(request))
val data: Data           = dbLookup(request)
futureAd.onComplete { ad =>
   val page = makePage(data, ad.get)
   connection.write(page)
}
```

Listing 26.3: Ad-fetching example with a callback on a future.

After the connection-handling thread completes the database lookup, it registers a callback action on the ad-fetching task, instead of waiting for the task to finish. The callback action extracts a customized ad from the `Try` value (assuming no error), assembles the data and ad into a page, and sends the page back as a reply as before. The key difference from Listing 26.2 is that no blocking is involved.

## 26.3 Higher-Order Functions on Futures

By using a callback to assemble and send the page, you avoid blocking, and thus eliminate the risk of deadlock. However, because the function argument of `onComplete` is an action, which is executed for its side effects, some of the earlier functional flavor is lost. Recall that in the full version of the server in Listing 25.2, the assembled page is also used in a statistics-keeping function. If a value produced as a future is needed in multiple places, handling these uses with callbacks can get complicated. If the value is used asynchronously, you may even need callbacks within callbacks, which are difficult to write and even more difficult to debug. A better solution would be to bring the non-blocking nature of callbacks into code that maintains a more functional style.

Before we revisit the ad-fetching example in Section 26.5, consider this callback-based function:

*Scala*
```scala
def multiplyAndWrite(futureString: Future[String], count: Int): Unit =
   futureString.onComplete {
      case Success(str) => write(str * count)
      case Failure(e)   => write(s"exception: ${e.getMessage}")
   }
```

Somewhere, an input string is being produced asynchronously. It is passed to function `multiplyAndWrite` as a future. This function uses a callback to repeat the string multiple times and to write the result—in Scala, `"A" * 3` is `"AAA"`. This approach requires that you specify in the callback action everything you want to do with the string `str * count`, which does not exist outside this callback. This is a source of possible complexity and loss of modularity. It could be more advantageous to replace `multiplyAndWrite` with a `multiply` function that somehow returns the string `str * count` and makes it available to any code that needs it.

Within this `multiply` function, however, the string to multiply may not yet be available—its computation could still be ongoing. You also cannot wait for it because you want to avoid blocking. Instead, you need to return the multiplied string itself as a future. Accordingly, the return type of `multiply` is not `String`, but rather `Future[String]`. The future to be returned can be created using a promise, as in Listing 25.4:

*Scala*
```scala
def multiply(futureString: Future[String], count: Int): Future[String] =
   val promise = Promise[String]()
   futureString.onComplete {
      case Success(str) => promise.success(str * count)
      case Failure(e)   => promise.failure(e)
   }
   promise.future
```

You create a promise to hold the multiplied string and a callback action to fulfill the promise. If the future `futureString` produces a string, the callback multiplies it and fulfills the promise successfully. Otherwise, the promise is failed, since no string was available for multiplication.

Now comes the interesting part. Conceptually, the preceding code has little to do with strings and multiplication. What it really does is transform the value produced by a future so as to create a new future. Of course, we have seen this pattern before—for instance, to apply a function to the contents of an option—in the form of the higher-order function `map`. Instead of focusing on the special case of string multiplication, you could write a generic `map` function on futures:

*Scala*
```scala
def map[A, B](future: Future[A], f: A => B): Future[B] =
   val promise = Promise[B]()
   future.onComplete {
      case Success(value) => promise.complete(Try(f(value)))
      case Failure(e)     => promise.failure(e)
   }
   promise.future
```

This function is defined for generic types `A` and `B` instead of strings. The only meaningful difference with function `multiply` is that `f` might fail and is invoked inside `Try`. Consequently, the promise could be failed for one of two reasons: No value of type `A` is produced on which to apply `f`, or the invocation of function `f` itself fails.

The beauty of bringing up `map` is that it takes us to a familiar world, that of higher-order functions, as discussed in Chapters 9 and 10 and throughout Part I. Indeed, the `Try` type itself has a method `map`, which you can use to simplify the implementation of `map` on futures:

*Scala*
```scala
def map[A, B](future: Future[A], f: A => B): Future[B] =
   val promise = Promise[B]()
   future.onComplete(tryValue => promise.complete(tryValue.map(f)))
   promise.future
```

Listing 26.4: Reimplementing higher-order function `map` on futures.

Note how error cases are handled transparently, thanks to the `Try` type (see Section 13.3 for the behavior of `map` on `Try`).

Functions as values and higher-order functions constitute a fundamental aspect of functional programming. Chapter 25 introduced futures as a way to start making concurrent programming more functional by relying on a mechanism adapted to value-producing tasks. Once you choose to be functional, you should not be surprised to see higher-order patterns begin to emerge. But the story doesn't end with `map`. Various higher-order functions on futures give rise to a powerful concurrent programming style that is both functional and non-blocking.

You don't need to reimplement `map` on futures: Scala futures already have a `map` method. You can simply write function `multiply` as follows:

*Scala*
```scala
def multiply(futureString: Future[String], count: Int): Future[String] =
   futureString.map(str => str * count)
```

A thread that calls `multiply` does not block to wait for the input string to become available. Nor does it create any new string itself. It only makes sure that the input string will be multiplied once it is ready, typically by a worker from a thread pool.

## 26.4 Function flatMap on Futures

In function `multiply`, the string argument is given asynchronously, as a future, but the count is already known at call time and is passed as an integer. In a more general variant, the multiplying count itself could be the result of an asynchronous computation and be passed to `multiply` as a future. In that case, you need to combine `futureString`, of type `Future[String]`, and `futureCount`, of type `Future[Int]`, into a `Future[String]`. The expression

```
futureString.map(str => futureCount.map(count => str * count))
```

would have type `Future[Future[String]]`, which is not what you want. Of course, we have had this discussion before (with options, in Section 10.3) and used it to introduce the fundamental operation `flatMap`. Scala futures also have a `flatMap` method:

*Scala*
```scala
def multiply(futureString: Future[String], futureCount: Future[Int]): Future[String] =
  futureString.flatMap(str => futureCount.map(count => str * count))
```

There is nothing blocking in this function. Once both futures—`futureString` and `futureCount`—are completed, the new string will be created.

You can use other functions to achieve the same purpose. For instance, you can combine the two futures into a `Future[(String,Int)]` using `zip`, then use `map` to transform the pair:

*Scala*
```scala
def multiply(futureString: Future[String], futureCount: Future[Int]): Future[String] =
  futureString.zip(futureCount).map((str, count) => str * count)
```

You can even use `zipWith`, which combines `zip` and `map` into a single method:

*Scala*
```scala
def multiply(futureString: Future[String], futureCount: Future[Int]): Future[String] =
  futureString.zipWith(futureCount)((str, count) => str * count)
```

The last two functions may be easier to read than the `flatMap/map` variant. Nevertheless, you should keep in mind the fundamental nature of `flatMap`. Indeed, `zip` and `zipWith` can be implemented using `flatMap`.

An experienced Scala programmer might write `multiply` as follows:

*Scala*
```scala
def multiply(futureString: Future[String], futureCount: Future[Int]): Future[String] =
   for str <- futureString; count <- futureCount yield str * count
```

Recall from Section 10.9 that `for-yield` in Scala is implemented as a combination of `map` and `flatMap` (and `withFilter`). This code would be transformed by the compiler into the earlier `flatMap/map` version. The `for-yield` syntax is very nice, especially when working with futures. I encourage you to use it if you are programming in Scala. However, as mentioned in an earlier note, I will continue to favor the explicit use of `map` and `flatMap` in this book's examples for pedagogical reasons.

By combining two futures into one—using `flatMap`, `zip`, or `zipWith`—you can rewrite the parallel quick-sort example of Listing 26.1 as a non-blocking function:

*Scala*
```scala
def quickSort(list: List[Int])(using ExecutionContext): Future[List[Int]] = list match
   case Nil => Future.successful(list)
   case pivot :: others =>
      val (low, high) = others.partition(_ < pivot)
      val lowFuture   = Future.delegate(quickSort(low))
      val highFuture  = quickSort(high)
      lowFuture.flatMap(lowSorted =>
         highFuture.map(highSorted => lowSorted ::: pivot :: highSorted)
      )
```

Listing 26.5: Non-blocking implementation of parallel quick-sort.

To avoid blocking, the return type of the function is changed from `List[Int]` to `Future[List[Int]]`. As before, the task of sorting the low values is delegated to the thread pool. You could equivalently write it as `lowFuture = Future(quickSort(low)) .flatten`. A direct recursive call is used to sort the high values, and the two futures are combined, following the same pattern as in function `multiply`. This variant of quick-sort involves no blocking and, in contrast to Listing 26.1, cannot possibly deadlock.[1]

[1]This implementation remains inefficient. Its main weakness is that it creates separate sorting tasks for very small lists down to the empty list. More realistic implementations would stop the parallelization at some point and sort short lists within the current thread instead. Java's `Arrays.parallelSort` function, for instance, stops distributing subarrays to separate threads once they have 8192 or fewer elements.

## 26.5 Illustration: Parallel Server Revisited

Equipped with standard higher-order functions on futures, we can now go back to the server example. First, you can replace the callback action in Listing 26.3 with a call to `map` to produce a `Future[Page]` out of a `Future[Ad]`:

*Scala*

```scala
val futureAd: Future[Ad]     = Future(fetchAd(request))
val data: Data               = dbLookup(request)
val futurePage: Future[Page] = futureAd.map(ad => makePage(data, ad))
futurePage.foreach(page => connection.write(page))
```

Listing 26.6: Ad-fetching example with `map` and `foreach` on futures.

You use `map` to transform an ad into a full page by combining the ad with data already retrieved from the database. You now have a `Future[Page]`, which you can use wherever the page is needed. In particular, sending the page back to the client is a no-value action, for which a callback fits naturally. For illustration purposes, this code registers the callback with `foreach` instead of `onComplete`. The two methods differ in that `foreach` does not deal with errors: Its action is not run if the future fails.

In Listing 26.6, the database is queried by the connection-handling thread while a customized ad is fetched in the background. Alternatively, database lookup can be turned over to another thread, resulting in a value of type `Future[Data]`. You can then combine the two futures using `flatMap/map`:

*Scala*

```scala
val futureAd: Future[Ad]     = Future(fetchAd(request))
val futureData: Future[Data] = Future(dbLookup(request))
val futurePage: Future[Page] =
   futureData.flatMap(data => futureAd.map(ad => makePage(data, ad)))
futurePage.foreach(page => connection.write(page))
```

Listing 26.7: Ad-fetching example with `flatMap` and `foreach` on futures.

What is interesting about this code is that the thread that executes it does not perform any database lookup, ad fetching, or page assembling and writing. It simply creates futures and invokes non-blocking higher-order methods on them. If you write the remainder of the connection-handling code in the same style, you end up with a `handleConnection` function that is entirely asynchronous and non-blocking:

*Scala*

```scala
given exec: ExecutionContextExecutorService =
   ExecutionContext.fromExecutorService(Executors.newFixedThreadPool(16))
```

```
val server = ServerSocket(port)

def handleConnection(connection: Connection): Unit =
   val requestF = Future(connection.read())
   val adF      = requestF.map(request => fetchAd(request))
   val dataF    = requestF.map(request => dbLookup(request))
   val pageF    = dataF.flatMap(data => adF.map(ad => makePage(data, ad)))
   dataF.foreach(data => addToLog(data))
   pageF.foreach(page => updateStats(page))
   pageF.foreach(page => { connection.write(page); connection.close() })

while true do handleConnection(Connection(server.accept()))
```

Listing 26.8: A fully non-blocking parallel server.

The `handleConnection` function starts by submitting to the thread pool a task that reads a request from a socket and produces a future, `requestF`. From then on, the code proceeds by calling higher-order functions on futures. First, using `map`, an ad-fetching task is scheduled to run once the request has been read. This call produces a future `adF`. A database lookup future, `dataF`, is created in the same way. The two futures `dataF` and `adF` are combined into a future `pageF` using `flatMap` and `map`, as before. Finally, three callback actions are registered: one on `dataF` for logging, and two on `pageF` for statistics recording and to reply to the client.

No actual connection-handling work is performed by the thread that runs function `handleConnection`. The thread simply creates futures and invokes non-blocking functions on them. The time it takes to run the entire body of `handleConnection` is negligible. In particular, you could make the listening thread itself do it, in contrast to Listing 25.2, where a separate task is created for this purpose.

The various computations that need to happen when handling a request depend on each other, as depicted in Figure 26.2. The server implemented in Listing 26.8 executes a task as soon as its dependencies have been completed, unless all 16 threads in the pool are busy. Indeed, the 16 threads jump from computation to computation—fetching ads, logging, building pages, and so on—as the tasks become eligible to run, across request boundaries. They never block, unless there is no task at all to run. This implementation maximizes parallelism and is deadlock-free.

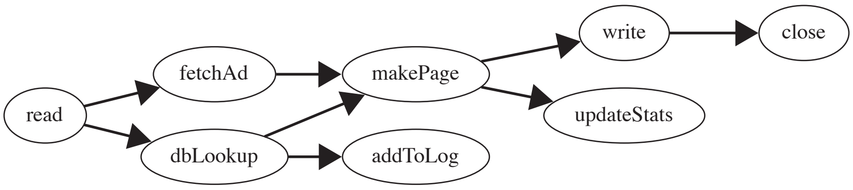

**Figure 26.2 Activity dependencies in the server example.**

Instead of Scala futures, you could implement the server of Listing 26.8 using Java's `CompletableFuture` (see Appendix A.14 for a pure Java implementation). Note, however, that `CompletableFuture` tends to use less standard names: `thenApply`, `thenCompose`, and `thenAccept` are equivalent to `map`, `flatMap`, and `foreach`, respectively.

There is one aspect of concurrent programming that a non-blocking approach tends to make more difficult: handling timeouts. For instance, you could decide that it is undesirable to have the server wait for more than 0.5 second for a customized ad after data has been retrieved from the database. In a blocking style, you can achieve this easily by adding a timeout argument when invoking `futureAd.get`, as in Listing 25.3. It can be somewhat more challenging when using a non-blocking style.

Here, `CompletableFuture` has the advantage over Scala's `Future`. It defines a method `completeOnTimeout` to complete a future with an alternative value after a given timeout. If the future is already finished, `completeOnTimeout` has no effect. You can use it to fetch a default ad:

*Scala*
```scala
val adF: CompletableFuture[Ad] = ...
...
adF.completeOnTimeout(timeoutAd, 500, MILLISECONDS)
```

Scala's `Future` type has no such method, which makes the implementation of a timeout ad more difficult. You can follow a do-it-yourself approach by creating a promise and relying on an external timer to complete it if needed. First, you create a timer as a scheduling thread pool:

*Scala*
```scala
val timer = Executors.newScheduledThreadPool(1)
```

Then, you create a promise when the database lookup finishes:

*Scala*
```scala
val pageF = dataF.flatMap { data =>
   val safeAdF =
      if adF.isCompleted then adF
      else
         val promise = Promise[Ad]()
         val timerF =
            timer.schedule(() => promise.trySuccess(timeoutAd), 500, MILLISECONDS)
         adF.foreach { ad =>
            timerF.cancel(false)
            promise.trySuccess(ad)
         }
         promise.future
   safeAdF.map(ad => makePage(data, ad))
}
```

This code runs when future `dataF` completes—when data from the database becomes available. If, at that point, the ad is ready—`adF.isCompleted` is true—you can use it. Otherwise, you need to make sure that an ad will be available quickly. For this purpose, you create a promise, and you use the timer to fulfill this promise with a default ad after 0.5 second. You also add a callback action to `adF`, which tries to fulfill the same promise. Whichever runs first—the timer task or the customized ad task—will set the promise with its value.[2] The call `timerF.cancel` is not strictly necessary, but it is used to avoid creating an unnecessary default ad if the customized ad is available in time.

As part of the last case study, Listing 28.4 uses a similar strategy to extend Scala futures with a `completeOnTimeout` method.

## 26.6 Functional-Concurrent Programming Patterns

Both futures, on the one hand, and higher-order functions, on the other hand, are powerful abstractions. Together, they form a potent combination, though one that can take some effort to master. Even so, it is a worthwhile effort. This section illustrates a few guidelines you should keep in mind as you venture into functional-concurrent programming.

### flatMap as an Alternative to Blocking

Higher-order functions are abstractions for code you don't have to write. They are convenient but could often be replaced with handwritten implementations. If `opt` is an option, for instance, `opt.map(f)` could also be written:

*Scala*
```scala
opt match
   case Some(value) => Some(f(value))
   case None        => None
```

In the case of futures, however, higher-order functions are an alternative to computations that would be hard to implement directly. If `fut` is a future, what can you replace `fut.map(f)` with? A future cannot simply be "opened" to access its value, since the value may not yet exist. Short of creating—and blocking—additional threads, there is no alternative to using higher-order functions to act inside a future.

You can leverage your functional programming skills with higher-order functions when working with futures. Earlier, for instance, we used `flatMap` on options to chain computations that may or may not produce a value. You can use `flatMap` in a similar way on futures to chain computations that may or may not be asynchronous. Instead of "optional" stages, from `A` to `Option[B]`, you define asynchronous stages, as functions from `A` to `Future[B]`.

[2]Method `trySuccess` is used because method `success` fails with an exception when invoked on an already completed promise.

As an illustration, the three optional functions used in Section 10.3 can be changed to represent asynchronous steps:

Scala
```
def parseRequest(request: Request): Future[User] = ...
def getAccount(user: User): Future[Account] = ...
def applyOperation(account: Account, op: Operation): Future[Int] = ...
```

The steps can then be chained using `flatMap`:

Scala
```
parseRequest(request)
   .flatMap(user => getAccount(user))
   .flatMap(account => applyOperation(account, op))
```

Listing 26.9: A pipeline of futures using `flatMap`.

The expression in Listing 26.9 is *exactly* the same as that in Listing 10.5, except that it produces a value of type `Future[Int]` instead of `Option[Int]`.

## Uniform Treatment of Synchronous and Asynchronous Computations

You could mix synchronous and asynchronous operations by combining steps of type `A => B`—using `map`—and steps of type `A => Future[B]`—using `flatMap`. Instead, it is often more convenient to use only steps of the form `A => Future[B]` combined with `flatMap`. When needed, synchronous steps can be implemented as already completed futures. This design increases flexibility: It makes it easier to replace synchronous steps with asynchronous steps, and vice versa.

For instance, if accounts are simply stored in a map, the `getAccount` function from the earlier example can be implemented synchronously, within the calling thread:

Scala
```
val allAccounts: Map[User, Account] = ...
def getAccount(user: User): Future[Account] = Future.successful(allAccounts(user))
```

This function returns an already completed future and does not involve any additional thread. If a need to fetch accounts asynchronously then arises, you can reimplement the function without modifying its signature, and leave all the code that uses it—such as Listing 26.9—unchanged.

## Functional Handling of Failures

Exceptions are typically thrown and caught within a thread. They don't naturally travel from thread to thread, and they are ill suited for multithreaded programming. Instead, you are better off following the functional approach to error handling discussed in Chapter 13.

An added benefit of relying on computations of type `A => Future[B]` instead of `A => B` is that futures can also carry failures—in Scala, you can think of `Future` as an asynchronous `Try`. For example, you can improve the `getAccount` function by making sure it always produces a future, even when a user is not found:

*Scala*
```scala
def getAccount(user: User): Future[Account] = Future.fromTry(Try(allAccounts(user)))
```

This way, an expression like `getAccount(user).onComplete(...)` still executes a callback action, which is not true if `getAccount` throws an exception. Failed futures can be handled functionally, using dedicated functions such as `recover` in Scala or `exceptionally` in Java.

For simplicity, the connection-handling function from Listing 26.8 does not deal with errors. You could use standard future functions to add robustness to the server. For instance, failure to create a page could be handled by transforming the `pageF` future:

*Scala*
```scala
val safePageF: Future[Page] = pageF.recover { case ex: PageException => errorPage(ex) }
```

or by adding a failure callback:

*Scala*
```scala
pageF.failed.foreach { ex =>
   connection.write(errorPage(ex))
   connection.close()
}
```

Either the callback actions specified using `pageF.foreach` or those specified using `pageF.failed.foreach` will run, but not both.

## Non-Blocking "Join" Pattern

In the server example, `pageF` is created by combining two futures, `dataF` and `adF`, using `flatMap`. You can use the same approach to combine three or more futures:

*Scala*
```scala
val f1: Future[Int]    = ...
val f2: Future[String] = ...
val f3: Future[Double] = ...

val f: Future[(Int, String, Double)] =
   f1.flatMap(n => f2.flatMap(s => f3.map(d => (n, s, d))))
```

This won't scale to larger numbers of futures, though. An interesting and not uncommon case is to combine $N$ futures of the same type into a single one, for an arbitrary number $N$. In the server example, a client might obtain data from $N$ database queries, which are executed in parallel:

```scala
def queryDB(requests: List[Request]): Future[Page] =
   val futures: List[Future[Data]] = requests.map(request => Future(dbLookup(request)))
   val dataListF: Future[List[Data]] = Future.sequence(futures)
   dataListF.map(makeBigPage)
```

The first line uses `map` to create a list of database-querying tasks, one for each request. These tasks, which run in parallel, form a list of futures. The key step in `queryDB` is the call to `Future.sequence`. This function uses an input of type `List[Future[A]]` to produce an output of type `Future[List[A]]`. The future it returns is completed when all the input futures are completed, and it contains all their values as a list (assuming no errors). Invoking `Future.sequence` serves the same purpose as the "join" part of a fork-join pattern, but does so without blocking. The last step uses a function `makeBigPage` from `List[Data]` to `Page` to build the final page.

As of this writing, there is no standard `sequence` function for `CompletableFuture`, but you can implement your own using `thenCompose` (equivalent to `flatMap`) and `thenApply` (equivalent to `map`):

```scala
def sequence[A](futures: List[CompletableFuture[A]]): CompletableFuture[List[A]] =
   futures match
      case Nil => CompletableFuture.completedFuture(List.empty)
      case future :: more =>
         future.thenCompose(first => sequence(more).thenApply(others => first::others))
```

Listing 26.10: Joining a list of `CompletableFutures` into one without blocking.

This function uses recursion to nest calls to `thenCompose` (`flatMap`). In the recursive branch, `sequence(more)` is a future that will contain the values of all the input futures, except the first. This future and the first input future are then combined using `thenCompose` and `thenApply` (`flatMap` and `map`), according to the pattern used earlier to merge two futures (as in Listings 26.5, 26.7, and 26.8).

## Non-Blocking "Fork-Join" Pattern

Function `queryDB` uses a fork-join pattern in which `sequence` implements the "join" part without blocking. Fork-join is a common enough pattern that Scala defines a function `traverse` that implements both the "fork" and "join" parts of a computation. You can use it for a simpler implementation of `queryDB`:

```scala
def queryDB(requests: List[Request]): Future[Page] =
   Future.traverse(requests)(request => Future(dbLookup(request))).map(makeBigPage)
```

Instead of working on a list of futures, as `sequence` does, function `traverse` uses a list of inputs and a function from input to future output. It "forks" a collection of tasks by applying the function to all inputs, and then "joins" the tasks into a single future, as in `sequence`, without blocking.

## 26.7 Summary

- When used as a synchronizer, a future requires a thread to potentially block and wait to access the value being computed. Parking and unparking threads has a non-negligible performance cost. Worse yet, tasks that block to wait on other tasks running on the same group of threads can easily result in deadlocks. It is not always possible to avoid these deadlocks by increasing the number of threads in a pool, and larger pool sizes tend to cause inefficiencies even when it can be done.
- Callbacks can be used as an alternative to blocking. They trigger a computation when a future is ready, without having to explicitly wait for this future to finish. Callbacks can be complex—future values can be used in arbitrary ways—and can lead to intricate code, especially when callbacks within callbacks are involved.
- By defining non-blocking higher-order functions on futures, you can bring to the world of concurrent programming the same shift from actions to functions that is at the core of functional programming. Instead of using effect-based callbacks, future values are handled functionally, as when using functions, but asynchronously.
- The resulting functional-concurrent programming style does not use futures as synchronizers—thereby avoiding many deadlock scenarios and performance costs associated with blocking—and also sidesteps the inherent complexity of callbacks.
- Higher-order functions on futures can be used to transform values, combine multiple computations asynchronously, or recover from failures. The same higher-order functions that proved hugely beneficial to functional programming—particularly, `map`, `flatMap`, `foreach`, and `filter`—provide developers with tools to orchestrate complex concurrent computations according to patterns that maximize concurrency while avoiding blocking.
- Functions `flatMap` and `map`, in particular, can be used to combine in a uniform way computations that may be synchronous or asynchronous, failed or successful. They can also be used to implement, without blocking threads, patterns that (conceptually) wait for tasks to finish, such as fork-join.
- Adjusting to functional-concurrent programming requires a shift in program design, away from locks and synchronizers. This can initially require some effort, similar to casting aside assignments and loops when moving from imperative to functional programming. Once you become accustomed to it, though, this programming style is often easier and less error-prone than the alternatives.

# Index